Introduction to Programming in Python

Strings

Dr. Bill Young
Department of Computer Science
University of Texas at Austin

Last updated: June 4, 2021 at 11:04

Strings and Characters

A **string** is a sequence of characters. Python treats strings and characters in the same way. Use either single or double quote marks.

```
letter = 'A'        # same as letter = "A"
numChar = "4"       # same as numChar = '4'
msg = "Good morning"
```

(Many) characters are represented in memory by binary strings in the ASCII (American Standard Code for Information Interchange) encoding.

Strings and Characters

A string is represented in memory by a sequence of ASCII character codes. So manipulating characters really means manipulating these numbers in memory.

...	...	
...	...	
2000	01001010	Encoding for character 'J'
2001	01100001	Encoding for character 'a'
2002	01110110	Encoding for character 'v'
2003	01100001	Encoding for character 'a'
...	...	
...	...	

ASCII

The following is part of the ASCII (American Standard Code for Information Interchange) representation for characters.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
32		!	”	#	$	%	&	’	(	)	*	+	,	-	.	/
48	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
64	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
96	‘	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
112	p	q	r	s	t	u	v	w	x	y	z	{	—	}		

The standard ASCII table defines 128 character codes (from 0 to 127), of which, the first 32 are control codes (non-printable), and the remaining 96 character codes are representable characters.

Unicode

ASCII codes are only 7 bits (some are extended to 8 bits). 7 bits only allows 128 characters. *There are many more characters than that in the world.*

Unicode is an extension to ASCII that uses multiple bytes for character encodings. With Unicode you can have Chinese characters, Hebrew characters, Greek characters, etc.

Unicode was defined such that ASCII is a subset. So Unicode readers recognize ASCII.

Operating on Characters

Notice that:

- The lowercase letters have consecutive ASCII values (97...122); so do the uppercase letters (65...90).
- The uppercase letters have lower ASCII values than the uppercase letters, so "less" alphabetically.
- There is a difference of 32 between any lowercase letter and the corresponding uppercase letter.

To convert from upper to lower, add 32 to the ASCII value.

To convert from lower to upper, subtract 32 from the ASCII value.

To sort characters/strings, sort their ASCII representations.

ord and chr

Two useful functions for characters:

- `ord(c)` : give the ASCII code for character `c`; returns a number.
- `chr(n)` : give the character with ASCII code `n`; returns a character.

```
>>> ord('a')
97
>>> ord('A')
65
>>> diff = (ord('a') - ord('A'))
>>> diff
32
>>> upper = 'R'
>>> lower = chr( ord(upper) + diff ) # upper to lower
>>> lower
'r'
>>> lower = 'm'
>>> upper = chr( ord(lower) - diff ) # lower to upper
>>> upper
'M'
```

Escape Characters

Some special characters wouldn't be easy to include in strings, e.g., single or double quotes.

```
>>> print("He said: "Hello"")
  File "<stdin>", line 1
    print("He said: "Hello"")
                           ^
SyntaxError: invalid syntax
```

What went wrong?

To include these in a string, we need an *escape sequence*.

Escape Sequence	Name	Escape Sequence	Name
\n	linefeed	\'	single quote
\f	formfeed	\"	double quote
\b	backspace	\r	carriage return
\t	tab	\\	backslash

Creating Strings

Strings are immutable meaning that two instances of the same string are really the same object.

```
>>> s1 = str("Hello")    # using the constructor function
>>> s2 = "Hello"         # alternative syntax
>>> s3 = str("Hello")
>>> s1 is s2             # are these the same object?
True
>>> s2 is s3
True
```

Functions on Strings

Some functions that are available on strings:

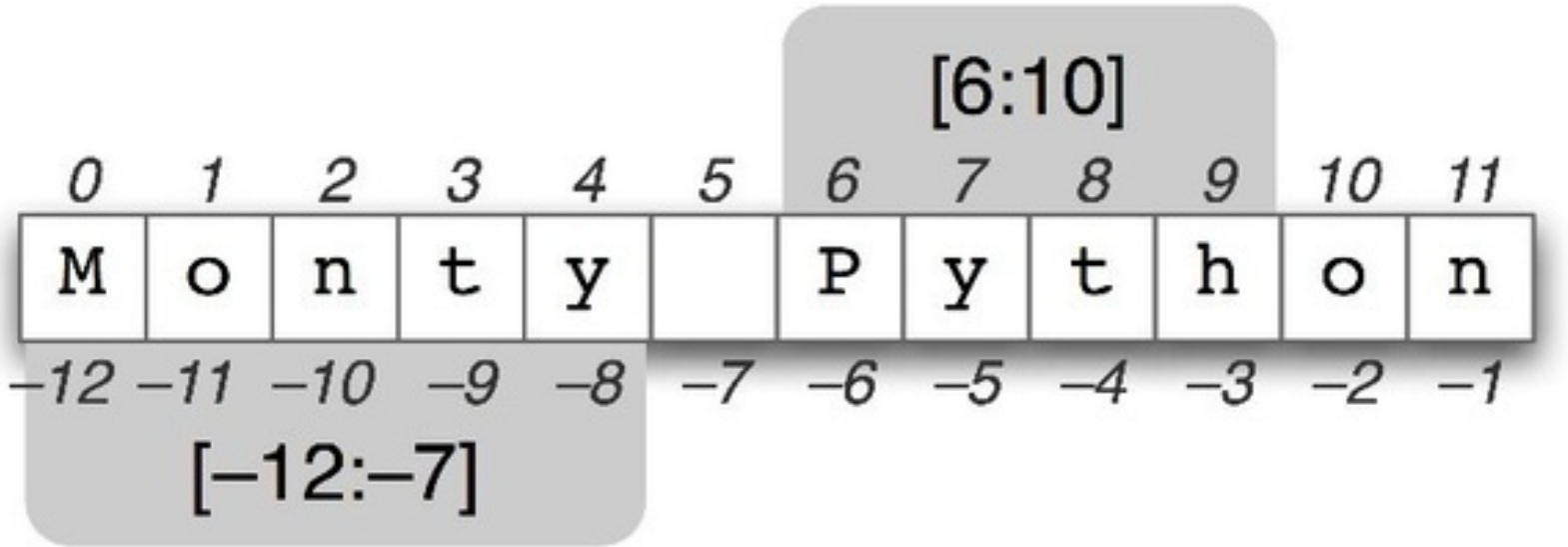
Function	Description
`len(s)`	return length of the string
`min(s)`	return char in string with lowest ASCII value
`max(s)`	return char in string with highest ASCII value

```
>>> s1 = "Hello, World!"
>>> len(s1)
13
>>> min(s1)
' '
>>> min("Hello")
'H'
>>> max(s1)
'r'
```

Why does it make sense for a blank to have lower ASCII value than any letter?

Indexing into Strings

Strings are sequences of characters, which can be accessed via an index.



Indexes are 0-based, ranging from [0 ... len(s)-1].

You can also index using negatives, `s[-i]` means `-i+len(s)]`.

Indexing into Strings

```
>>> s = "Hello, World!"
>>> s[0]
'H'
>>> s[6]
' '
>>> s[-1]
'!'
>>> s[-6]
'W'
>>> s[-6 + len(s)]
'W'
```

Slicing

Slicing means to select a contiguous subsequence of a sequence or string.

General Form:

```
String[start : end]
```

```
>>> s = "Hello, World!"
>>> s[1 : 4]                    # substring from s[1]...s[3]
'ell'
>>> s[ : 4]                     # substring from s[0]...s[3]
'Hell'
>>> s[1 : -3]                   # substring from s[1]...s[-4]
'ello, Wor'
>>> s[1 : ]                     # same as s[1 : s(len)]
'ello, World!'
>>> s[ : 5]                     # same as s[0 : 5]
'Hello'
>>> s[:]                        # same as s
'Hello, World!'
>>> s[3 : 1]                    # empty slice
''
```

Concatenation and Repetition

General Forms:

```
s1 + s2
s * n
n * s
```

`s1 + s1` means to create a new string of `s1` followed by `s2`.
`s * n` or `n * s` means to create a new string containing `n` repetitions of `s`

```
>>> s1 = "Hello"
>>> s2 = ", World!"
>>> s1 + s2                 # + is not commutative
'Hello, World!'
>>> s1 * 3                  # * is commutative
'HelloHelloHello'
>>> 3 * s1
'HelloHelloHello'
```

Notice that concatenation and repetition *overload* two familiar operators.

`in` and `not in` operators

The `in` and `not in` operators allow checking whether one string is a *contiguous* substring of another.

General Forms:

```
s1 in s2
s1 not in s2
```

```
>>> s1 = "xyz"
>>> s2 = "abcxyzrls"
>>> s3 = "axbyczd"
>>> s1 in s2
True
>>> s1 in s3
False
>>> s1 not in s2
False
>>> s1 not in s3
True
```

Comparing Strings

In addition to equality comparisons, you can order strings using the relational operators: `<`, `<=`, `>`, `>=`.

For strings, this is *lexicographic* (or alphabetical) ordering using the ASCII character codes.

```
>>> "abc" < "abcd"
True
>>> "abcd" <= "abc"
False
>>> "Paul Jones" < "Paul Smith"
True
>>> "Paul Smith" < "Paul Smithson"
True
>>> "Paula Smith" < "Paul Smith"
False
```

Iterating Over a String

Sometimes it is useful to do something to each character in a string, e.g., change the case (lower to upper and upper to lower).

```
DIFF = ord('a') - ord('A')

def swapCase (s):
    result = ""
    for ch in s:
        if ( 'A' <= ch <= 'Z' ):
            result += chr(ord(ch) + DIFF )
        elif ( 'a' <= ch <= 'z' ):
            result += chr(ord(ch) - DIFF )
        else:
            result += ch
    return result

print(swapCase( "abCDefGH" ))
```

```
> python StringIterate.py
ABcdEFgh
```

Strings are Immutable

You can't change a string, by assigning at an index. You have to create a new string.

```
>>> s = "Pat"
>>> s[0] = 'R'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> s2 = 'R' + s[1:]
>>> s2
'Rat'
```

Whenever you concatenate two strings or append something to a string, you create a new value.

Functions vs. Methods

Python is an Object Oriented Language; everthing data item is a member of a **class**. For example, integers are members of class `int`.

When you type `2 + 3`, that's really syntactic shorthand for `int.__add__(2, 3)`, calling method `__add__` on the class `int` with arguments 2 and 3.

When you call `len( lst )`, that's really shorthand for `lst.__len__()`.

General form:

```
item.method( args )
```

So many things that look like function calls in Python are really method invocations. That's not true of functions you write.

Useful Testing Methods

You have to get used to the syntax of method invocation.

Below are some useful methods on strings. *Notice that they are methods, not functions, so called on string s.*

Function	Description
`s.isalnum()`:	nonempty alphanumeric string?
`s.isalpha()`:	nonempty alphabetic string?
`s.isdigit()`:	nonempty and contains only digits?
`s.isidentifier()`:	follows rules for Python identifier?
`s.islower()`:	nonempty and contains only lowercase letters?
`s.isupper()`:	nonempty and contains only uppercase letters?
`s.isspace()`:	nonempty and contains only whitespace?

Useful Testing Methods

```
>>> s1 = "abc123"
>>> isalpha( s1 )                  # wrong syntax
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'isalpha' is not defined
>>> s1.isalpha()
False
>>> "1234".isdigit()
True
>>> "abCD".isupper()
False
>>> "\n\t   \b".isspace()
False
>>> "\n\t   \t".isspace()
True
```

Substring Search

Python provides some string methods to see if a string contains another as a substring:

Function	Description
`s.endswith(s1):`	does s end with substring s1?
`s.startswith(s1):`	does s start with substring s1?
`s.find(s1):`	lowest index where s1 starts in s, -1 if not found
`s.rfind(s1):`	highest index where s1 starts in s, -1 if not found
`s.count(s1):`	number of non-overlapping occurrences of s1 in s

Substring Search

```
>>> s = "Hello, World!"
>>> s.endswith("d!")
True
>>> s.startswith("hello")          # case matters
False
>>> s.startswith("Hello")
True
>>> s.find('l')                    # search from left
2
>>> s.rfind('l')                   # search from right
10
>>> s.count('l')
3
>>> "abababababa".count('aba')     # nonoverlapping occurrences
2
```

String Exercise

The string `count` method counts nonoverlapping occurrences of one string within another.

```
>>> "abababababa".count('aba')
2
>>> "abababababa".count('c')
0
```

Suppose we wanted to write a function that would count *all* occurrences, including possibly overlapping ones.

String Exercise

In file countOverlaps.py:

```
def countOverlaps( txt, s ):
    """ Count the occurrences of s in txt,
    including possible overlapping occurrences. """
    count = 0
    while len(txt) >= len(s):
        if txt.startswith(s):
            count += 1
        txt = txt[1:]
    return count
```

Running our code:

```
>>> from countOverlaps import *
>>> txt = "abababababa"
>>> s = "aba"
>>> countOverlaps(txt, s)
5
>>>
```

Converting Strings

Below are some additional methods on strings. Remember that strings are *immutable*, so these all make a new copy of the string.

Function	Description
`s.capitalize()`:	return a copy with first character capitalized
`s.lower()`:	lowercase all letters
`s.upper()`:	uppercase all letters
`s.title()`:	capitalize all words
`s.swapcase()`:	lowercase letters to upper, and vice versa
`s.replace(old, new)`:	replace occurences of old with new

String Conversions

```
>>> "abcDEfg".upper()
'ABCDEFG'
>>> "abcDEfg".lower()
'abcdefg'
>>> "abc123".upper()            # only changes letters
'ABC123'
>>> "abcDEF".capitalize()
'Abcdef'
>>> "abcDEF".swapcase()         # only changes letters
'ABCdef'
>>> book = "introduction to programming using python"
>>> book.title()                # doesn't change book
'Introduction To Programming Using Python'
>>> book2 = book.replace("ming", "s")
>>> book2
'introduction to programs using python'
>>> book2.title()
'Introduction To Programs Using Python'
>>> book2.title().replace("Using", "With")
'Introduction To Programs With Python'
```

Stripping Whitespace

It's often useful to remove whitespace at the start, end, or both of string input. Use these functions:

Function	Description
`s.lstrip()`:	return copy with leading whitespace removed
`s.rstrip()`:	return copy with trailing whitespace removed
`s.strip()`:	return copy with leading and trailing whitespace removed

```
>>> s1 = "    abc    "
>>> s1.lstrip()                    # new string
'abc    '
>>> s1.rstrip()                    # new string
'    abc'
>>> s1.strip()                     # new string
'abc'
>>> "a b c".strip()
'a b c'
```

String Exercise

Exercise: Input a string from the user. Count and print out the number of lower case, upper case, and non-letters.

String Exercise

Exercise: Input a string from the user. Count and print out the number of lower case, upper case, and non-letters.

In file `CountCases.py`:

```
def countCases( txt ):
    """ For a text, count and return the number of lower
    upper, and non-letter letters. """
    lowers = 0
    uppers = 0
    nonletters = 0
    # For each character in the text, see if lower, upper,
    # or non-letter and increment the count.
    for ch in txt:
        if ch.islower():
            lowers += 1
        elif ch.isupper():
            uppers += 1
        else:
            nonletters += 1
    # Return a triple of the counts.
    return lowers, uppers, nonletters
```

Calling countCases

```
def main():
    txt = input("Please enter a text: ")
    lc, uc, nl = countCases( txt )
    print("Contains:")
    print("   Lower case letters:", lc)
    print("   Upper case letters:", uc)
    print("   Non-letters:", nl)

main()
```

Here's a sample run:

```
> python CountCases.py
Please enter a text: abcXYZ784*&^def
Contains:
   Lower case letters: 6
   Upper case letters: 3
   Non-letters: 6
```